Chapter 1

Background & State of the art

This chapter discusses state of the art in the areas of *WebAssembly*, *Diversification* and *Runtime Randomization*. We present a summary of the relevant related work and the key concepts and background that we use along with this writing. We select the discussed works by their novelty and critical insights to provide automatic diversification.

TODO Description of Wasm section

1.1 WebAssembly overview

Over the past decades, JavaScript has been used in the majority of the browser clients to allow client-side scripting. However, due to the complexity of this language and to gain in performance, several approaches appeared, supporting different languages in the browser. For example, Java applets were introduced on web pages late in the 90's, Microsoft made an attempt with ActiveX in 1996 and Adobe added ActionScript later on 1998. All these attempts failed to persist, mainly due to security issues and the lack of consensus on the community of browser vendors.

In 2014, Emscripten tried with a strict subset of JavaScript, asm.js, to allow low level code such as C to be compiled to JavaScript itself. Asm.js was first implemented as an LLVM backend. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [33] compared to standard JavaScript code. The main reason why asm.js is faster, is that it limits the language features to those that can be optimized in the LLVM pipeline or those that can be directly translated from the source code. Besides, it removes the majority of the dynamic characteristics of the language, limiting it to numerical types, top-level functions, and one large array in the memory directly accessed as raw data. Since asm.js is a subset of JavaScript it was compatible with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standarization.

The work of Van Es et al. [24] proposed to shrink JavaScript to asm.js in a source-to-source strategy, closing the cycle and extending the fact that asm.js was mainly a compilation target for C/C++ code. Despite their result was encouraging, JavaScript faces several limitations related to the intrinsics of the language. For example, any JavaScript engine requires the parsing and the recompilation of the JavaScript code creating overhead.

We, as many in the community consider asm.js the precursor of WebAssembly. The WebAssembly (Wasm) language was first publicly announced in 2015. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially stated later by the work of Haas et al. [22] in 2017. The announcement of WebAssembly marked the first step of standarizing bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it outperforms asm.js [22]. Since then, the adoption of WebAssembly has an exponential growth. Cases like the following evidence this fact; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a fully Minecraft version [2]; and the case of Blazor [3], a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

From source to Wasm

All WebAssembly programs are compiled ahead of time from source languages. LLVM includes Wasm as a target since version 8, supporting a broad range of frontend languages such as C/C++. The resulting binary, works similarly to a traditional shared library, it includes instruction codes, symbols and exported functions. In Figure 1.1, we illustrate the workflow of the creation of Wasm binaries and their lifecycle. The process starts by compiling the source code program to Wasm (Step ①). This step includes ahead-of-time optimizations. To the best of our knowledge, the most complete survey about Wasm tooling is presented in the Awesome Wasm (`https://github.com/mbasso/awesome-wasm`) repo. It is a cumulative GitHub repo which includes references to articles, papers, books, demos, compilers and engines related to WebAssembly.

The step ② includes the composition of the glue code to Wasm as JavaScript code. This code will include the harness that the Wasm binary needs for its execution. For example, the functions to interact with the DOM of the HTML page are imported in the Wasm binary during its call from the JavaScript glue code. The glue code can be manually written, however, compilers like Emscripten, Rust and Binaryen can generate it automatically, making this process completely transparent to developers.

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2] `https://satoshinm.github.io/NetCraft/`

[3] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

Finally, the third step (Step ③), includes the compilation and execution of the client-side code. Most of the browser engines compile either the Wasm and JavaScript codes to machine code. In the case of JavaScript, this process involves JIT and hot code replacement during runtime. For Wasm, this process is pseudo-direct. For instance, in the case of V8, the engine used in Chrome, only two compilation processes are required, a baseline compilation, meant to make the coming binary available as soon as possible and a final one, that makes lane and fast optimizations such as constant folding and dead code removal. Once V8 completes the second compilation process, the generated machine code for Wasm is final and is the same used along all its executions. This analysis was validated by conversations with the V8's dev team and by experimental studies in our previous contributions.

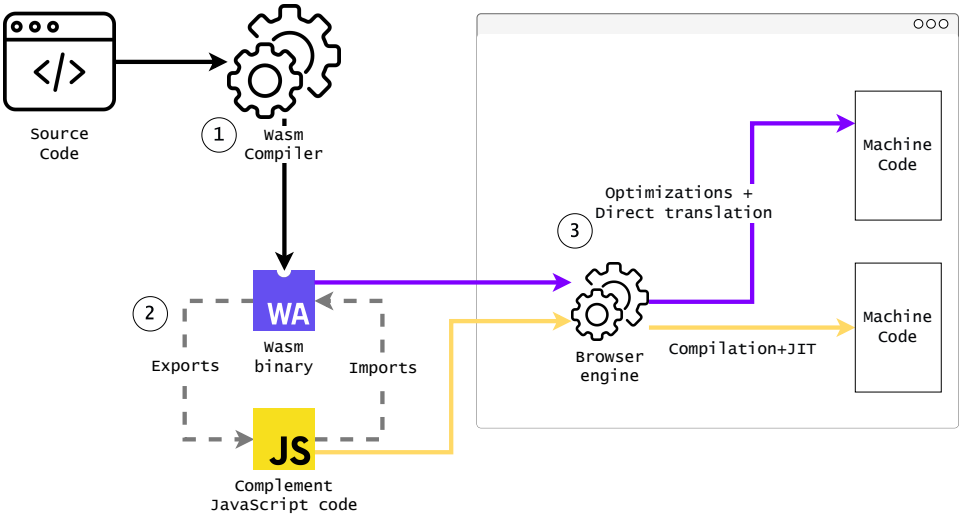

Figure 1.1: WebAssembly building and lifecycle.

The Wasm binary might not need external glue code. Thus, depends on the host environment and the binary itself to supply it. This fact encourages the adoption of WebAssembly further browser engines. For instance, Cloudflare and Fastly adapted their platforms to provide Function as a Service (FaaS) directly with WebAssembly. In this case the glue code, instead of JavaScript, is provided by any other language stack that the host environment supports. In 2019, the bytecode alliance team [4] proposed WebAssembly System Interface (WASI). WASI is the foundation to build Wasm code outside of the browser with a POSIX system interface platform. It standarizes the adoption of WebAssembly outside web browsers [11] in heterogeneous platforms like the Edge or IoT devices [3, 9].

[4] `https://bytecodealliance.org/`

WebAssembly specificities

WebAssembly defines its own Instruction Set Architecture (ISA) [19]. It is an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types, each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

In Listing 1.1 and Listing 1.2 we illustrate an arbitrary C code and its compilation to Wasm respectively. The C function contains: heap allocation, external function declaration and the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in Listing 1.2 is in the textual format for the generated Wasm. The module in this case first defines the signature of the functions(Line 2, Line 3 and Line 4) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (Line 5, Line 35 and Line 36). The definition of the function (Line 6) and its body follows the last import declaration at Line 5.

The function body is composed by local variable declarations and typed instructions that are evaluated in a virtual stack (Line 7 to Line 32 in Listing 1.2). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of Listing 1.2, the result value of the main function is the calculation of the last instruction, `i32.add` at Line 32. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm. As the listing shows, instructions are annotated with a type. Wasm binaries contain only four datatypes, `i32`, `i64`, `f32` and `f64`.

Wasm manages the memory in a restricted way.A Wasm module has a linear memory component that is accessed as `i32` pointers and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in Line 37. The memory access is illustrated in Line 15. This memory is usually bound in browser engines to 2Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in Line 33 and Line 36). Therefore, this ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (Line 34) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (Line 30) or using a static table of function declarations. This latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, however, the compiler should populate the static table of functions.

Listing 1.1: Arbitrary C function.

```c
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 1.2: WebAssembly code for Listing 1.1.

```
1  (module
2    (type (;0;) (func (param f32) (result i32)))
3    (type (;1;) (func))
4    (type (;2;) (func (result i32)))
5    (import "env" "ftoi" (func $ftoi (type 0)))
6    (func $main (type 2) (result i32)
7      (local i32 i32)
8      i32.const -1000
9      local.set 0 ;loop iteration counter;
10     block  ;label = @1;
11       loop  ;label = @2;
12         i32.const 0
13         local.get 0
14         i32.add
15         i32.load
16         local.tee 1
17         i32.const 101
18         i32.ge_s ;loop iteration condition;
19         br_if 1 ;@1;
20         local.get 0
21         i32.const 4
22         i32.add
23         local.tee 0
24         br_if 0 ;@2;
25       end
26       i32.const 0
27       return
28     end
29     f32.const 0x1.9147aep+3 ;=12.54;
30     call $ftoi
31     local.get 1
32     i32.add)
33   (memory (;0;) 1)
34   (global (;4;) i32 (i32.const 1000))
35   (export "memory" (memory 0))
36   (export "A" (global 2))
37   (data $data (0) "\00\00\00\00...")
38 )
```

Wasm control flow structures are different to standard assembly programs. All instructions are grouped into blocks, being the start of a function the root block. Two consecutive block declarations can be appreciated in Line 10 and Line 11 of Listing 1.2. Control flow structures jump between block boundaries and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (Line 25 and Line 28). By design, each block executes independently and cannot execute or refer to outer block values. This is guarantied by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (Line 19 and Line 24) and table break. Each breaking block instruction can only jump the execution to one of its enclosing blocks. For example, in Listing 1.2,

Line 19 will force the execution to jump to the end of the first block at Line 10 if the value at the top of the stack is greater than zero.

We want to remark that de description of Wasm in this section follows the version 1.0 of the language and not its proposals for extended features. We follow those features implemented in the majority of the vendors according to the Wasm roadmap [20]. On the other hand we excluded instructions for datatype conversion, table accesses and the majority of the arithmetic instructions for the sake of simplicity.

WebAssembly's security

As we described, WebAssembly is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [10] and makes easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

However, WebAssembly is vulnerable under certain conditions, at the execution engine's level [15]. Implementations in both browsers and standalone runtimes [3] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [17]. One of our previous contributions trigger a CVE on the code generation component of wasmtime, highlighting that even when the language specification is meant to be secure, its lifecycle might not. Moreover, Also, binaries itself can be vulnerable. The work of Lehmann et al. [8] proved the moving of C/C++ source code vulnerabilities to Wasm such as overwriting constant data or manipulating the heap using stackoverflow. Even though this vulnerabilities need a specific glue code to be exploited, they make a call for better defenses for WebAssembly. Several proposals for extending WebAssembly in the current roadmap could address some existing vulnerabilities. For example, having multiple memories could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors.

We see in WebAssembly a monoculture problem, if one environment is vulnerable, all the others are vulnerable in the same manner as the same WebAssembly binary is replicated. On the other hand, the WebAssembly environment lacks natural diversity [32]. Compared to the work of Harrand et al. [?] , in WebAssembly, one could not use preexisting and different program versions to provide diversification for monoculture solving. In fact, according to the work of Hilbig et al. [5], the artificial variants created with one of our works contribute to the half of executable and available WebAssembly binaries in the wild. The current limitations on security and the lack of preexisting diversity motivate our work on software diversification as one preemptive mitigation among a wide

range of security countermeasures.

1.2 Diversification and Superdiversification

TODO Draw landscape. Emphasis on why for security, sidechannels, etc.
TODO Check nicos How to in background related to diversity.

Program diversification approaches can be applied at different stages of the development pipeline. This section analyzes the related works for both static and dynamic diversification. Besides, we motivate superoptimization strategies to provide a "superdiversifier" that uses intermediate solutions to search for optimal programs to provide program variants. Finally, we describe our contribution to the field.

TODO Searching for software diversity: attaining artificial diversity through program synthesis

Static diversification consists in synthesizing, building, and distributing different, functionally equivalent binaries to end-users. This aims at increasing the complexity and applicability of an attack against a large population of users [54]. **TODO** Take a look to Building diverse computer systems Dealing with code-reuse attacks, Homescu et al. [35] propose inserting NOP instruction directly in LLVM IR to generate a variant with a different code layout at each compilation. In this area, Coppens et al. [36] use compiler transformations to iteratively diversify software. Their work aims to prevent reverse engineering of security patches for attackers targeting vulnerable programs. Their approach continuously applies a random selection of predefined transformations using a binary diffing tool as feedback [?] . A downside of their method is that attackers are, in theory, able to identify the type of transformations applied and find a way to ignore or reverse them.

Previous works have attempted to generate diversified variants that are alternated during execution. It has been shown to drastically increase the number of execution traces required by a side-channel attack. Amarilli et al. [41] is the first to propose the generation of code variants against side-channel attacks. Agosta et al. [34] and Crane et al. [31] modify the LLVM toolchain to compile multiple functionally equivalent variants to randomize the control flow of software, while Couroussé et al. [26] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks.

Jackson et al. [39] have explored how to use NOP operations inserted during compiling time to statically diversify programs. Another idea **TODO** Jackson is to use the optimization flags of several compilers to generate semantically equivalent binaries out of the same source code. These techniques place the compiler at the core of the diversification technique. However, this approach is limited by the number of available flags in the compiler implementation and because the optimization is applied in all possible places in the code at the same time.

Superoptimization

The search for optimal algorithms to compute a function is as older as of the first compiler. This problem is commonly solved by using human-written heuristics inside the compiler implementations. However, this solution has limitations. First, the optimizations are applied to small pieces of code and do not consider more complex processes like instruction selections, register allocation, and target-dependent optimizations. Second, the well-known phase ordering problem [38]. To solve this problem, Massalin et al. [55] proposed a superoptimizer, a statistical method to exhaustively explore all possible program constructions to find the smallest program. Given an input program, code superoptimization focuses on *searching* for a new program variant that is faster or smaller than the original code while preserving its correctness [27]. The search space for the optimal program is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by length in ascending order. If any of these programs are found to perform the same function as the source program, the search halts. However, the exhaustive exploration approach becomes virtually impossible for larger instruction sets. Because of this, the paper proposes a pruning method over the search space and a fast probabilistic test to check programs' equivalence.

Apart from recent works in the area of Machine Learning [2], to the best of our knowledge, there are two main implementations for superoptimizers using two completely different strategies. Churchill et al. [23] implement STOKE to superoptimize large programs for the Google Native Client stack. They use a bounded verifier to ensure that every generated optimization goes through all the checks for semantic equivalence. STOKE uses a probabilistic approach, following a Monte-Carlo-Markov-Chain strategy to select code transformations that lead to smaller programs. On the other hand, Souper [48] automatically generates smaller programs for LLVM following an exhaustive enumerative synthesis. Souper finds subexpressions at the LLVM function level and builds all possible expressions from all the instructions that are no larger than the original subexpression. When Souper finds a replacement, it uses an SMT solver [44] to verify the semantic equivalence with the original program. Superoptimization is more expensive than traditional optimization heuristics in compilers yet, provides more profound and more robust code transformations.

Superdiversification and statement of novelty

While finding optimized code, the idea and the implementations of superoptimization discard intermediate solutions that are semantically equivalent to the original program. The discarding of intermediate solutions follows the principle of optimization, finding the best possible program. Jacob et al. [43] propose the use of a "superdiversification" technique, inspired by superoptimization, to synthesize individualized versions of programs, their main idea is to keep the intermediate

solutions finding the optimal program. The tool developed by Jacob et al. does not output only the optimal instruction sequence but any semantically equivalent sequences. Their work focuses on a specific subset of x86 instructions.

In this research, we contribute to the state of the art in artificially creating diversity. While the number of related work for software diversity is enormous, no approach has been applied to the context of WebAssembly. One of our contributions, CROW, extrapolates the idea of superdiversification for WebAssembly. CROW works directly with LLVM IR, enabling it to generalize to more languages and CPU architectures, something not possible with the x86-specific approach of previous works. Furthermore, we conducted a sanity check for diversification preservation, researching to what extent browser compilers do not remove our introduced diversity.

CROW focuses on the static diversification of software. However, because of the specificities of code execution in the browser, this is not far from being a dynamic approach. For example, since WebAssembly is served at each page refreshment, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW. It also can be used in fuzzing campaigns [?] to provide reliability. The diversification created by CROW can unleash hidden behaviors in compilers and interpreters. By generating several functionally equivalent and yet different variants, deeper bugs can be discovered. Thanks to CROW, a bug was discovered in the Lucet compiler [5]. Fastly acknowledged our work as part of a technical blog post [6] that describes the bug and the patch.

1.3 Runtime diversification

In this section, we highlight past works on runtime strategy for diversification. Besides, we describe and discuss the foundation that supports the composition of diverse yet semantically equivalent programs to enforce security. Finally, we describe our contribution to the field.

A randomization technique creates a set of unique executions for the very same program [51]. Seminal works include instruction-set randomization [50, 52] to create a unique mapping between artificial CPU instructions and real ones. This makes it very hard for an attacker to ignore the key to inject executable code. This breaks the predictability of program execution and mitigates certain exploits.

Chew, and Song [53] target operating system randomization. They randomize the interface between the operating system and the user applications: the system call numbers, the library entry points (memory addresses), and the stack placement. All those techniques are dynamic, done at runtime using load-time preprocessing and rewriting. Bathkar et al. [51, 49] proposed three kinds of randomization transformations: randomizing the base addresses of applications and libraries memory regions, random permutation of the order of variables and

[5] `REPO`

[6] `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`

routines, and the random introduction of random gaps between objects. Dynamic randomization can address different kinds of problems. In particular, it mitigates an extensive range of memory error exploits. Recent work in this field includes stack layout randomization against data-oriented programming [14], and memory safety violations [4], as well as a technique to reduce the exposure time of persistent memory objects to increase the frequency of address randomization [7].

Moving Target Defense and Multivariant execution

Moving Target Defense (MTD) for software was first proposed as a collection of techniques that aim to improve the security of a system by constantly moving its vulnerable components [6]. Usually, MTD techniques revolve around changing system inputs and configurations to reduce attack surfaces. This increases uncertainty for attackers and makes their attacks more difficult. Ultimately, potential attackers cannot hit what they cannot see. MTD can be implemented in different ways, including via dynamic runtime platforms [12]. Segupta et al. illustrated how a dynamic MTD system [21] can be applied to different technology stacks. Using this technique, the authors illustrated that some CVE related to specific database engines could be avoided.

On the same topic, Multivariant Execution (MVE) can be seen as a Moving Target Defense strategy. In 2006, security researchers at University of Virginia laid the foundations of a novel approach to security that consists in executing multiple variants of the same program. They called this "N-variant systems" [47]. Bruschi et al. [46] and Salamat et al. [45] pioneered the idea of executing the variants in parallel. Subsequent techniques focus on Multivariant Execution (MVE) for mitigating memory vulnerabilities [16] and other specific security problems incl. return-oriented programming attacks [28] and code injection [40].A key design decision of MVE is whether it is achieved in kernel space [13], in user-space [42], with exploiting hardware features [25], or even through code polymorphism [18]. Finally, one can neatly exploit the limit case of executing only two variants [37**?** , 29]. Notably, Davi et al. proposed Isomeron [30], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. With this strategy a potential attacker cannot predict whether the original or the variant of a program will execute.

Statement of novelty

Researching on MVE in a distributed setting like the Edge [?] has been less researched. Voulimeneas et al. proposed a multivariant execution system by parallelizing the execution of the variants in different machines [1] for the sake of efficiency. Since CROW offers static and runtime diversity for WebAssembly

and its adoption for the Edge and backend executions are becoming security-sensitive fields, we propose an original kind of MVE, MEWE. We generate multiple program variants, which we execute on edge computing nodes. We use the natural redundancy of Edge-Cloud computing architectures to deploy an internet-based MVE.

With MEWE, We contribute to the field of randomization at two stages. First, we automatically generate variants of a given program with CROW, which have different WebAssembly code and still behave the same. Second, we randomly select which variant is executed at runtime, creating a multivariant execution scheme that randomizes the observable behaviors at each run of the program.

1.4 Conclusions

Software Diversification has been widely researched, not being the case in the WebAssembly context. With this dissertation, we aim to settle down the foundation to study automatic diversification for WebAssembly. We contribute to the field of artificial diversity by extending the superdiversifier idea of Jacob et al. [43]. We empirically demonstrate that CROW provides robust program diversification. Finally, we propose a novel approach of merging program variants to provide multivariant execution. Our contributions are obtained by following the methodology described in **??**.

Bibliography

[1] Voulimeneas,A., Song,D., Larsen,P., Franz,M., and Volckaert,S. (2021). dmvx: Secure and efficient multi-variant execution in a distributed setting. In *Proceedings of the 14th European Workshop on Systems Security*, pages 41–47.

[2] Shypula,A., Yin,P., Lacomis,J., Le Goues,C., Schwartz,E., and Neubig,G. (2021). Learning to Superoptimize Real-world Programs. *arXiv e-prints*, page arXiv:2109.13498.

[3] Narayan,S., Disselkoen,C., Moghimi,D., Cauligi,S., Johnson,E., Gang,Z., Vahldiek-Oberwagner,A., Sahita,R., Shacham,H., Tullsen,D., et al. (2021). Swivel: Hardening webassembly against spectre. In *USENIX Security Symposium*.

[4] Lee,S., Kang,H., Jang,J., and Kang,B. B. (2021). Savior: Thwarting stack-based memory safety violations by randomizing stack layout. *IEEE Transactions on Dependable and Secure Computing*.

[5] Hilbig,A., Lehmann,D., and Pradel,M. (2021). An empirical study of real-world webassembly binaries: Security, languages, use cases. *Proceedings of the Web Conference 2021*.

[6] (2021). National Cyber Leap Year.

[7] Xu,Y., Solihin,Y., and Shen,X. (2020). Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proc. of ASPLOS*, pages 987–1000.

[8] Lehmann,D., Kinder,J., and Pradel,M. (2020). Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[9] Gadepalli,P. K., McBride,S., Peach,G., Cherkasova,L., and Parmer,G. (2020). Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, page 265–279.

[10] Chen,D. and W3C group (2020). WebAssembly documentation: Security. Accessed: 18 June 2020.

[11] Bryant,D. (2020). Webassembly outside the browser: A new foundation for pervasive computing. In *Proc. of ICWE 2020*, pages 9–12.

[12] Roy,A., Chhabra,A., Kamhoua,C. A., and Mohapatra,P. (2019). A moving target defense against adversarial machine learning. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, page 383–388.

[13] Österlund,S., Koning,K., Olivier,P., Barbalace,A., Bos,H., and Giuffrida,C. (2019). kmvx: Detecting kernel information leaks with multi-variant execution. In *ASPLOS*.

[14] Aga,M. T. and Austin,T. (2019). Smokestack: thwarting dop attacks with runtime stack layout randomization. In *Proc. of CGO*, pages 26–36.

[15] Silvanovich,N. (2018). The problems and promise of webassembly. Technical report.

[16] Lu,K., Xu,M., Song,C., Kim,T., and Lee,W. (2018). Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing*.

[17] Genkin,D., Pachmanov,L., Tromer,E., and Yarom,Y. (2018). Drive-by key-extraction cache attacks from portable code. *IACR Cryptol. ePrint Arch.*, 2018:119.

[18] Belleville,N., Couroussé,D., Heydemann,K., and Charles,H.-P. (2018). Automated software protection for the masses against side-channel attacks. *ACM Trans. Archit. Code Optim.*, 15(4).

[19] WebAssembly Community Group (2017b). WebAssembly Specification.

[20] WebAssembly Community Group (2017a). WebAssembly Roadmap.

[21] Sengupta,S., Vadlamudi,S. G., Kambhampati,S., Doupé,A., Zhao,Z., Taguinod,M., and Ahn,G.-J. (2017). A game theoretic approach to strategy generation for moving target defense in web applications. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, page 178–186.

[22] Haas,A., Rossberg,A., Schuff,D. L., Schuff,D. L., Titzer,B. L., Holman,M., Gohman,D., Wagner,L., Zakai,A., and Bastien,J. F. (2017). Bringing the web up to speed with webassembly. *PLDI*.

[23] Churchill,B., Sharma,R., Bastien,J., and Aiken,A. (2017). Sound loop superoptimization for google native client. *SIGPLAN Not.*, 52(4):313–326.

[24] Van Es,N., Nicolay,J., Stievenart,Q., D'Hondt,T., and De Roover,C. (2016). A performant scheme interpreter in asm.js. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 1944–1951, New York, NY, USA. Association for Computing Machinery.

[25] Koning,K., Bos,H., and Giuffrida,C. (2016). Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE.

[26] Couroussé,D., Barry,T., Robisson,B., Jaillon,P., Potin,O., and Lanet,J.-L. (2016). Runtime code polymorphism as a protection against side channel attacks. In *IFIP International Conference on Information Security Theory and Practice*, pages 136–152. Springer.

[27] Bunel,R., Desmaison,A., Pawan Kumar,M., Torr,P. H., and Kohli,P. (2016). Learning to superoptimize programs. *arXiv e-prints*, 1(1):arXiv:1611.01787.

[28] Volckaert,S., Coppens,B., and De Sutter,B. (2015). Cloning your gadgets: Complete rop attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4).

[29] Kim,D., Kwon,Y., Sumner,W. N., Zhang,X., and Xu,D. (2015). Dual execution for on the fly fine grained execution comparison. *SIGPLAN Not.*

[30] Davi,L., Liebchen,C., Sadeghi,A.-R., Snow,K. Z., and Monrose,F. (2015). Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*.

[31] Crane,S., Homescu,A., Brunthaler,S., Larsen,P., and Franz,M. (2015). Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, pages 8–11.

[32] Baudry,B. and Monperrus,M. (2015). The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Comput. Surv.*, 48(1).

[33] Alon Zakai (2015). asm.js Speedups Everywhere.

[34] Agosta,G., Barenghi,A., Pelosi,G., and Scandale,M. (2015). The MEET approach: Securing cryptographic embedded software against side channel attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1320–1333.

[35] Homescu,A., Neisius,S., Larsen,P., Brunthaler,S., and Franz,M. (2013). Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE.

[36] Coppens,B., De Sutter,B., and Maebe,J. (2013). Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–26.

[37] Maurer,M. and Brumley,D. (2012). Tachyon: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 617–630.

[38] Kulkarni,S. and Cavazos,J. (2012). Mitigating the compiler optimization phase-ordering problem using machine learning. *SIGPLAN Not.*, 47(10):147–162.

[39] Jackson,T. (2012). *On the Design, Implications, and Effects of Implementing Software Diversity for Security*. PhD thesis, University of California, Irvine.

[40] Salamat,B., Jackson,T., Wagner,G., Wimmer,C., and Franz,M. (2011). Runtime defense against code injection attacks using replicated execution. *IEEE Trans. Dependable Secur. Comput.*, 8(4):588–601.

[41] Amarilli,A., Müller,S., Naccache,D., Page,D., Rauzy,P., and Tunstall,M. (2011). Can code polymorphism limit information leakage? In *IFIP International Workshop on Information Security Theory and Practices*, pages 1–21. Springer.

[42] Salamat,B., Jackson,T., Gal,A., and Franz,M. (2009). Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46.

[43] Jacob,M., Jakubowski,M. H., Naldurg,P., Saw,C. W. N., and Venkatesan,R. (2008). The superdiversifier: Peephole individualization for software protection. In *International Workshop on Security*, pages 100–120. Springer.

[44] de Moura,L. and Bjørner,N. (2008). Z3: An efficient smt solver. In Ramakrishnan,C. R. and Rehof,J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

[45] Salamat,B., Gal,A., Jackson,T., Manivannan,K., Wagner,G., and Franz,M. (2007). Stopping buffer overflow attacks at run-time: Simultaneous multi-variant program execution on a multicore processor. Technical report, Technical Report 07-13, School of Information and Computer Sciences, UCIrvine.

[46] Bruschi,D., Cavallaro,L., and Lanzi,A. (2007). Diversified process replicæ for defeating memory error exploits. In *Proc. of the Int. Performance, Computing, and Communications Conference*.

[47] Cox,B., Evans,D., Filipi,A., Rowanhill,J., Hu,W., Davidson,J., Knight,J., Nguyen-Tuong,A., and Hiser,J. (2006). N-variant systems: a secretless framework for security through diversity. In *Proc. of USENIX Security Symposium*, USENIX-SS'06.

[48] Bansal,S. and Aiken,A. (2006). Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 394–403, New York, NY, USA. Association for Computing Machinery.

[49] Bhatkar,S., Sekar,R., and DuVarney,D. C. (2005). Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the USENIX Security Symposium*, pages 271–286.

[50] Kc,G. S., Keromytis,A. D., and Prevelakis,V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proc. of CCS*, pages 272–280.

[51] Bhatkar,S., DuVarney,D. C., and Sekar,R. (2003). Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium*.

[52] Barrantes,E. G., Ackley,D. H., Forrest,S., Palmer,T. S., Stefanovic,D., and Zovi,D. D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. CCS*, pages 281–289.

[53] Chew,M. and Song,D. (2002). Mitigating buffer overflows by operating system randomization. Technical Report CS-02-197, Carnegie Mellon University.

[54] Cohen,F. B. (1993). Operating system protection through program evolution. *Computers & Security*, 12(6):565–584.

[55] Henry,M. (1987). Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126.